HP Operations Analytics

Software Version: 2.31

AQL Developer Guide

Document Release Date: September 2015
Software Release Date: September 2015

Legal Notices

Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

Trademark Notices

Adobe™ is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark of The Open Group.

Documentation Updates

The title page of this document contains the following identifying information:

- Software Version number, which indicates the software version.
- Document Release Date, which changes each time the document is updated.
- Software Release Date, which indicates the release date of this version of the software.

To check for recent updates or to verify that you are using the most recent edition of a document, go to: https://softwaresupport.hp.com/group/softwaresupport/search-result?keyword=.

This site requires an HP Passport account. If you do not have one, click the **Create an account** button on the HP Passport Sign in page.

Support

Visit the HP Software Support web site at: **https://softwaresupport.hp.com**

This web site provides contact information and details about the products, services, and support that HP Software offers.

HP Software Support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valued support customer, you can benefit by using the support web site to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Download software patches
- Manage support contracts
- Look up HP support contacts
- Review information about available services
- Enter into discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and sign in. Many also require a support contract. To register for an HP Passport ID, go to **https://softwaresupport.hp.com** and click **Register**.

To find more information about access levels, go to: **https://softwaresupport.hp.com/web/softwaresupport/access-levels**

HP Software Solutions & Integrations and Best Practices

Visit HP Software Solutions Now at **https://h20230.www2.hp.com/sc/solutions/index.jsp** to explore how the products in the HP Software catalog work together, exchange information, and solve business needs.

Visit the Cross Portfolio Best Practices Library at **https://hpln.hp.com/group/best-practices-hpsw** to access a wide variety of best practice documents and materials.

Contents

Chapter 1: What is AQL

Use the Analytics Query Language (AQL) when the Phrased Query Language (PQL) syntax is not specific enough to return the data you need. When using AQL you can be more specific about the data collected. You can also filter, group, and order the collected data in a single query.

The primary objective of the Analytics Query Language (AQL) is to simplify your ad hoc query experience. This applies to the process of building custom dashboards as well as troubleshooting problems using statistical algorithms.

AQL is a hierarchical language that provides layers of abstraction on analytic queries. The idea here is that the more abstract, the easier it is for you to write AQL in an ad hoc fashion. The layers of abstraction in AQL are:

- Built-in analytics are defined as functions that become intrinsic in AQL
- A query language to provide SQL-like access to all collections
- Functions and expressions as abstractions of queries

Note: This manual includes examples that show script usage, command line usage, command line syntax, and file editing. If you copy and paste any examples from this manual, carefully review the results of your paste before running a command or saving a file.

As an example of the layers of abstraction, consider the following query:

Note: When using AQL, you will be searching for collected metrics, such as `cpu_util`, which is shown in the following AQL query. Metrics are collected values over time for measurements such as system up time and CPU utilization.

```
from i in (oa_sysperf_global)
let interval=300
let analytic_interval=between($starttime,$endtime)
where (i.host_name like "myhost")
select moving_avg(i.cpu_util)
```

This query assumes a collection of system metrics from a predefined Operations Analytics collection (`oa_sysperf_global`) and will calculate a time series of the moving average of the CPU utilization for the system called `"myhost"`. The time series data is every 300 seconds (5 minutes) and the time range is specified by the internal macros `$starttime` and `$endtime`.

Note: The above example uses `moving_avg`, which is a built-in analytic that significantly simplifies this transformation over standard SQL.

It is clear that this query pattern is quite useful for all sorts of metrics. Suppose you have other metrics and other functions and want to calculate the time series of a particular metric using a particular function r for a particular host or hosts. You would use a query pattern as shown above.

Operations Analytics AQL supports using query patterns to be abstracted into AQL functions. Using the above example, suppose you want to generalize the query to generate a time series of any metric in `oa_sysperf_global` using any function for any set of hosts. To generate this time series, define an AQL function as follows:

```
/* Returns the moving analytic of a specific HP Operations Agent metric by host.
Input parameters are the host filter, metric name, and moving analytic function
name. */
define oaSysperfMovingMetric(hostFilter, metric, function) =
from i in (oa_sysperf_global)
let analytic_interval = between($starttime,$endtime)
let interval = $interval
where i.host_name like hostFilter
group by i.host_name
select function(i.metric)
```

With this function defined, the following AQL expression:

```
[oaSysperfMovingMetric("myhost", cpu_util, moving_avg)]
```

is identical to the above AQL query.

In addition, the following expression:

```
[oaSysperfMovingMetric("myhost", swap_util, moving_max)]
```

would give you the time series of the moving maximum swap utilization on host `"myhost"`.

The ability to define specialized AQL provides a significant 'ease-of-use' factor in using Operations Analytics to do ad hoc analytics. As further examples, Operations Analytics includes several packages of useful AQL functions that can be seen by using the Operations Analytics console.

Chapter 2: Using the Analytics Query Language (AQL)

AQL queries use a syntax similar to the ANSI Standard SQL. When using AQL, it is helpful if you have minimal knowledge of databases as well as scripting or programming skills. However, it is not mandatory to have this knowledge to get started using AQL queries.

Before you begin writing AQL queries, view the collection information that is stored in Operations Analytics to determine the kinds of data available in your environment. You will use this information as part of your AQL syntax. For details, see *How to View Collection Information* in the Operations Analytics help.

You can specify an AQL query, an AQL function, or an AQL expression when adding or editing a dashboard query pane. See *Dashboards and Query Panes* in the Operations Analytics help for more information.

Chapter 3: Analytics Query Language Syntax, Intrinsics, and Examples

AQL Syntax

The basic structure of an AQL query is very similar to the standard 'Structured Query Language'. An AQL query is a sequence of clauses. The clauses you include depend on the type, organization, and order of the information you want Operations Analytics to return. It also depends on the time range and type of analysis you want Operations Analytics to apply to the data.

The types of clauses supported by AQL are as follows:

- `from <row variable> in <collection>`
- `where <relational expression>`
- `let <name> = <value>`
- `group by <list of columns>`
- `select <select expression>`

When positioning the clauses in an AQL query, note the following:

1. The `from` and `select` clauses are mandatory. The `from` clause must be the first clause and the `select` clause must be the last clause in the query.
2. All other clauses in the query can be in any order between the `from` and the `select` clauses. The following clauses filter and group the identified collection of metrics and attributes.

> **Note:** An attribute is a descriptor for an entity, such as `host_name`, that is stored in a collection.

From Clause

The `from` clause defines the row variable and specifies the collection from which the rows will be selected. For example:

```
from i in (oa_sysperf_global)
```

defines the row variable to be i and the collection (table) to select from as `oa_sysperf_global`.

Where Clause

The `where` clause is any arbitrary relational expression. The `where` clause specifies the criteria for which rows are selected from the collection. The following is an example that shows on way to use the `where` clause to select rows by specific criteria:

```
where (( i.hostinfo_dnsname like "myhost")
&& (( i.severity ilike "CRITICAL" )||(i.severity ilike "WARNING") ))
```

This `where` clause restricts the selected rows to be only those events for host `"myhost"` with severity of either `CRITICAL` or `WARNING`.

Using special characters in a where clause

You can use special characters such as `{`, `}`, `[`,`]`, `\`, or `\\` embedded in a `where` clause filter condition strings. You can use the following methods:

- Specify a double backslash (`\\`) to represent an embedded literal backslash (`\`)
- Specify a backslash followed by double quote to represent an embedded literal double quote. For example, to represent a literal " use `\"`
- Specify a double backslash (\\) followed by a curly brace to represent a literal curly brace. For example, to represent a literal `{ or }`, use `\\{ or \\}`

The following example demonstrates the use of a double backslash to represent an embedded literal backslash. The intent of this example is to specify a filter such that host_names starting with `ab\c` are returned:

```
where (i.host_name like "ab\\c*")
```

The following example demonstrates the use of an escaped double quote (`\"`) to represent an embedded literal double quote (`"`). The intent of this example is to specify a filter such that host_names equaling `ab"c` are returned.

```
where (i.host_name == "ab\"c")
```

The following example demonstrates the use of double backslashes to represent embedded literal curly braces (`{ or }`). The intent of this example is to specify a filter such that host_names not equaling `ab {c}d` are returned.

```
where (i.host_name != "ab\\{c\\}d")
```

Let Clause

The `let` clause is used to define a value for a specific control variable for the query. For example, to control the time interval of the query, use the `let` clause to define a value for the global control variable `analytic interval` (for example, `analytic_interval=between($starttime, $endtime)` is where `$starttime` and `$endtime` are UI parameters.

The `let` clause can also be used to override dashboard pane parameters. For example it can override the limit setting that controls the number of results. The default for `Limit` is 100 and `let Limit = 50` would override the `Limit` dashboard pane parameter that is set to return just 50 results.

Group By

The `group by` clause organizes the results in the query based on the column or columns specified in the `group by` clause. For example `group by i.hostname` displays the results of the query in distinct groups by the host name attribute.

You can specify multiple columns in the `group by clause`, meaning the results will be organized primarily by the first column then by the second column, and so forth.

Note: You can specify multiple columns in the `group by` clause, meaning the results will be organized primarily by the first column then by the second column, and so forth.

Select Clause

The `select` clause explicitly specifies the values to be selected for the query results. If you specify just the row variable, all columns are selected by the query.

Examples:

```
select i
```

Selects all columns in the table.

```
select i.hostname, i.timestamp, i.state, i.category, i.title
```

Selects only the `hostname`, `timestamp`, `state`, `category`, and `title` attributes from the table.

Note: You can specify multiple columns in the `group by` clause, meaning the results will be organized primarily by the first column then by the second column, and so forth.

Intrinsic Statistical Functions in AQL

Operations Analytics provides a set of analytic functions to analyze the metrics, topology, inventory, event, and log file data that it collects.

Overall Aggregate (Summary) Functions Provided by Operations Analytics

The following table shows descriptions of the overall aggregate (summary) analytic functions provided by Operations Analytics.

Descriptions of Overall Aggregate (Summary) Functions Provided by Operations Analytics

Analytic Function Type	Description
`aggregate_avg`	Identifies the average value for the metric or metrics selected.
`aggregate_min`	Identifies the minimum value for the metric or metrics selected.
`aggregate_max`	Identifies the maximum value for the metric or metrics selected.
`aggregate_total`	Identifies the total value or cumulative sum for the metric or metrics selected.

Descriptions of Overall Aggregate (Summary) Functions Provided by Operations Analytics, continued

Analytic Function Type	Description
`aggregate_count`	Computes the total count of rows with values of an attribute or total count of all rows in a collection table.
`aggregate_ distinct_count`	Computes the total count of distinct values of an attribute.

Moving Aggregates (Time Series) Functions Provided by Operations Analytics

The following table shows descriptions of the moving aggregate (time series) functions provided by Operations Analytics.

Descriptions of Moving Aggregate (Time Series) Functions Provided by Operations Analytics

Function	Description
`moving_avg`	Computes the average values at each time interval within the specified time window for one or more metrics.
`moving_min`	Computes the minimum values at each time interval within the specified time window for one or more metrics.
`moving_max`	Computes the maximum values at each time interval within the specified time window for one or more metrics.
`moving_ total`	Computes the totals at each time interval within the specified time window for one or more metrics.
`moving_ count`	Computes the total counts of rows with values of an attribute or total count of all rows within a collection table at each time interval within the specified time window.
`moving_ distinct_ count`	Computes the total counts of distinct values of an attribute at each time interval within the specified time window.

Analytic Statistical Functions applied to Overall Aggregate and Moving Aggregate Functions

The following table describes the analytic statistical functions provided by Operations Analytics.

Descriptions of Analytic Statistical Functions applied to Overall Aggregate and Moving Aggregate Functions

Function	Description
`bottomN`	Computes the lowest N values in the expressions; returns the `bottomN` values with their associated rank.

Descriptions of Analytic Statistical Functions applied to Overall Aggregate and Moving Aggregate Functions, continued

Function	Description
`inverse_pctile`	Calculates the inverse percentile distribution values for the set of values in the expression. For example, if you specify 50 as the <*pctile*> value, `inverse_pctile` finds the 50th percentile value (or median value) for the data in the expression.
`pctile`	Calculates the percentile rank value for the values in the expressions. For example, if you specify 75 as the <*pctile*> value, `pctile` returns all values greater than the 75th percentile value for the data in the expression.
`rank`	Calculates the overall rank for all values in the expression, where the results include an integer (indicating rank) for each value along with the value itself.
`topN`	Uses the `rank` (descending order) analytic function to identify the highest N values. Operations Analytics returns the top N values with their associated rank. **Note:** • If you do not specify an N value in the AQL query, Operations Analytics displays the top five values. • The topN analytic function is not permitted in the `where` clause.

AQL Query Examples

Return the average CPU utilization and CPU run queue size

The following AQL query returns the average CPU utilization and CPU run queue size for each host matching the filter criteria.

```
from i in (oa_sysperf_global)
let analytic_interval= between($starttime,$endtime)
where (i.host_name like "*.mydomain.com") group by i.host_name
select aggregate_avg(i.cpu_util), aggregate_avg(i.cpu_run_queue)
```

Return the average for each of the metrics collected by the oa_sysperf_global collection

The following AQL query returns the average for each of the metrics collected by the `oa_sysperf_global` collection for each host matching the filter criteria:

```
from i in (oa_sysperf_global)
let analytic_interval= between($starttime,$endtime)
where (i.host_name like "*.mydomain.com") group by i.host_name
select aggregate_avg(i)
```

Return the maximum, minimum, and average values for CPU utilization and CPU run queue size

The following AQL query returns the maximum, minimum, and average for CPU utilization and CPU run queue size for each host matching the filter criteria:

```
from i in (oa_sysperf_global)
let analytic_interval= between($starttime,$endtime)
where (i.host_name like "*.mydomain.com") group by i.host_name
select aggregate_min(i.cpu_util),
aggregate_max(i.cpu_util),
aggregate_max(i.cpu_util),
aggregate_min(i.cpu_run_queue),
aggregate_max(i.cpu_run_queue),
aggregate_avg(i.cpu_run_queue)
```

Return the minimum, maximum, and average for each of the metrics collected by the oa_sysperf_global collection

The following AQL query returns the minimum, maximum and average for each of the metrics collected by the `oa_sysperf_global` collection for each host matching the filter criteria:

```
from i in (oa_sysperf_global)
let analytic_interval= between($starttime,$endtime)
where (i.host_name like "*.mydomain.com") group by i.host_name
select aggregate_min(i), aggregate_max(i), aggregate_avg(i)
```

Return Summary Information on Events (Example AQL Queries)

Note: Each of the examples queries data from the `omi_events_omievents` collection. This collection uses HP Operations Manager i (OMi) to collect OMi events. Each example queries data for only the hosts in the mydomain.com domain.

Return the total count of OMi events for a specified host and severity combination

The following AQL query calculates the total count of OMi events for each host and severity combinations matching the filter criteria: "

```
from i in (omi_events_omievents)
let analytic_interval= between($starttime, $endtime)
where ( ( i.hostinfo_dnsname like "*mydomain.com" ) && ( ( i.severity
ilike "CRITI*" ) || (i.severity
ilike "WARN*") ))
group by i.hostinfo_dnsname, i.severity select aggregate_count(i)
```

Return the total count of OMi events for a specified host and severity combination and for which the event count exceeds 100

The following AQL query does the same as the previous AQL query, except that it returns the counts for only those host name and severity combinations for which the event count exceeds 100:

```
from i in (omi_events_omievents)
let analytic_interval= between($starttime, $endtime)
where (( i.hostinfo_dnsname like "*mydomain.com" ) && ( ( i.severity ilike "CRITI*"
```

```
) || (i.severity
ilike "WARN*") ) && ( aggregate_count(i) > 100 ))
group by i.hostinfo_dnsname, i.severity select aggregate_count(i)
```

Return the number of distinct applications monitored by HP Business Process Monitor (BPM) per location

Note: The following AQL query uses the `bpm_application_performance` collection. This collection uses HP Business Process Monitor (BPM) to gather application performance information.

The following AQL query calculates the number of distinct applications monitored by BPM on a location by location basis:

```
from i in (bpm_application_performance)
let analytic_interval = between($starttime, $endtime)
group by i.location
select aggregate_distinct_count(i.application)
```

Return the total count of distinct database instances reporting Oracle metrics

Note: The following AQL query uses the `oa_oraperf_graph` collection. The `oa_oraperf_graph` collection uses HP Operations Smart Plug-in for Oracle to gather Oracle performance information.

The following AQL query returns a distinct count of database instances reporting Oracle metrics:

```
from i in (oa_oraperf_graph)
let analytic_interval= between($starttime,$endtime)
where ( i.host_name like "*mydomain.com" )
group by i.host_name select aggregate_distinct_count(i.db_instance_name)
```

Return the moving average CPU utilization and CPU run queue size

Note: Each of the examples queries data from the `oa_sysperf_global` collection. This collection uses HP Performance Agent to collect system metrics. Each example queries data for only the hosts in the mydomain.com domain.

The following AQL query returns the moving average CPU utilization and CPU run queue size for each host matching the filter criteria.

```
from i in (oa_sysperf_global)
let analytic_interval= between($starttime,$endtime) let interval=$interval
where (i.host_name like "*.mydomain.com") group by i.host_name
select moving_avg(i.cpu_util), moving_avg(i.cpu_run_queue)
```

Return the moving average for each of the metrics collected by the oa_sysperf_global collection

The following AQL query returns the moving average for each of the metrics collected by the `oa_sysperf_global` collection for each host matching the filter criteria:

```
from i in (oa_sysperf_global)
let analytic_interval= between($starttime,$endtime) let interval=$interval
where (i.host_name like "*.mydomain.com")
group by i.host_name
select moving_avg(i)
```

Return the moving maximum, minimum, and average values for CPU utilization and CPU run queue size

The following AQL query returns the moving maximum, minimum, and average for CPU utilization and CPU run queue size for each host matching the filter criteria:

```
from i in (oa_sysperf_global)
let analytic_interval= between($starttime,$endtime) let interval=$interval
where (i.host_name like "*.mydomain.com") group by i.host_name
select moving_min(i.cpu_util),
moving_max(i.cpu_util), moving_max(i.cpu_util), moving_min
(i.cpu_run_queue), moving_max(i.cpu_run_queue),
moving_avg(i.cpu_run_queue)
```

Return the moving minimum, maximum, and average for each of the metrics collected by the oa_sysperf_global collection

The following AQL query returns the moving minimum, maximum and average for each of the metrics collected by the `oa_sysperf_global` collection for each host matching the filter criteria:

```
from i in (oa_sysperf_global)
let analytic_interval= between($starttime,$endtime) let interval=$interval
where (i.host_name like "*.mydomain.com")
group by i.host_name
select moving_min(i), moving_max(i), moving_avg(i)
```

Note: Each of the following examples queries data from the `omi_events_omievents` collection. This collection uses HP Operations Manager i (OMi) to collect OMi events. Each example queries data for only the hosts in the mydomain.com domain.

Return the moving total count of OMi events for a specified host and severity combination

The following AQL query calculates the moving total count of OMi events for each host and severity combinations matching the filter criteria:

```
from i in (omi_events_omievents)
let analytic_interval=between($starttime,$endtime) let interval=$interval
where (( i.hostinfo_dnsname like "*mydomain.com" ) && ( ( i.severity ilike
"CRITI*" ) || (i.severity
ilike "WARN*") ))
group by i.hostinfo_dnsname, i.severity select moving_count(i)
```

Return the moving total count of OMi events for a specified host and severity combination and for which the event count exceeds 100

The following AQL query does the same as the previous AQL query, the difference being that it returns the moving counts for only those host name and severity combinations at only those intervals at which the event count exceeds 100:

```
from i in (omi_events_omievents)
let analytic_interval=between($starttime,$endtime) let interval=$interval
where (( i.hostinfo_dnsname like "*mydomain.com" ) && ( ( i.severity ilike
"CRITI*" ) || (i.severity
ilike "WARN*") ) && ( moving_count(i) > 100 ))
group by i.hostinfo_dnsname, i.severity select moving_count(i)
```

Return the moving number of distinct applications monitored by HP Business Process Monitor (BPM) per location.

Note: The following AQL query uses the `bpm_application_performance` collection. This collection uses HP Business Process Monitor (BPM) to gather application performance information.

The following AQL query calculates the moving number of distinct applications monitored by BPM on a location by location basis.

```
from i in (bpm_application_performance)
let analytic_interval = between($starttime, $endtime) let interval = $interval
group by i.location
select moving_distinct_count(i.application)
```

Return the moving total count of distinct database instances reporting Oracle metrics.

Note: The following AQL query uses the `oa_oraperf_graph` collection. The `oa_oraperf_graph` collection uses HP Operations Smart Plug-in for Oracle to gather Oracle performance information.

The following AQL query returns moving total counts of the distinct database instances reporting Oracle metrics:

```
from i in (oa_oraperf_graph)
let analytic_interval= between($starttime,$endtime) let interval = $interval where
( i.host_name like"*mydomain.com" )
group by i.host_name
select moving_distinct_count(i.db_instance_name)
```

Return the percentile distribution of overall cpu utilization by host

The following AQL query determines the hosts and their overall aggregate average values of CPU utilization along with the percentile rank for the value among the overall aggregate average values for all hosts matching the filter criteria:

```
from i in (oa_sysperf_global)
let analytic_interval= between($starttime,$endtime)
where( i.host_name like "*.mydomain.com" )
group by i.host_name select pctile(aggregate_avg(i.cpu_util) )
```

Note: The following example queries data from the `omi_events_omievents` collection. This collection uses HP Operations Manager i (OMi) to collect OMi events. Each example queries data for only the hosts in the mydomain.com domain.

Return the percentile distribution of event count by host

The following AQL query determines the hosts and their overall aggregate count of events along with percentile ranks of the overall aggregate event count values for all hosts matching the filter criteria:

```
from i in (omi_events_omievents)
let analytic_interval= between($starttime,$endtime)
where( i.hostinfo_dnsname like "*.mydomain.com" )
group by i.hostinfo_dnsname
select pctile(aggregate_count(i))
```

Note: The following example queries data from the `bpm_application_performance` collection. This collection uses HP Business Process Monitor (BPM) to gather application performance information.

Return the Top N Values (Example AQL Queries)

Tip: Also use these examples to assist you in constructing AQL queries that use the `bottomN` analytic function.

The following examples use the `topN` analytic function to return the top n values for sets of data returned by the overall aggregate and moving aggregate analytic functions.

Note: The following examples query data from the `oa_sysperf_global` collection. This collection uses HP Operations Agent to collect system metrics. Each example queries data for only the hosts in the mydomain.com domain.

Return the top five hosts and their overall aggregate average values of CPU utilization. This query also returns the associated relative ranks.

The following AQL query determines the top five hosts and their overall aggregate average values of CPU utilization among the overall aggregate average values and relative ranks for all hosts matching the filter criteria:

```
from i in (oa_sysperf_global)
let analytic_interval= between($starttime,$endtime)
where ( i.host_name like "*.mydomain.com" )
group by i.host_name
select topN(aggregate_avg(i.cpu_util),5 )
```

Return the top 10 hosts with the highest overall aggregate count of events

The following AQL query determines the top 10 hosts with the highest overall aggregate count of events among the overall aggregate event count values for all hosts matching the filter criteria:

```
from i in (omi_events_omievents)
let analytic_interval= between($starttime,$endtime)
```

```
where ( i.hostinfo_dnsname like "*.mydomain.com" )
group by i.hostinfo_dnsname
select topN(aggregate_count(i), 10 )
```

Chapter 4: Analytics Query Language Functions and Expressions

You have seen in the first section, "Analytics Query Language Syntax, Intrinsics, and Examples" on page 9, how to define an AQL function. This section explains how AQL functions are used and the style of AQL functions that are expected to be written.

Define Analytic Query Language Functions

By default, Operations Analytics provides several AQL functions to assist you with creating AQL queries, AQL functions, and associated dashboards. The concepts in this manual help you write your own AQL functions using a text editor.

You can write your own AQL functions using a text editor, then import these functions into Operations Analytics. Each text file you create can contain any number of AQL functions. A set of AQL functions that reside in a single file are known as an AQL module.

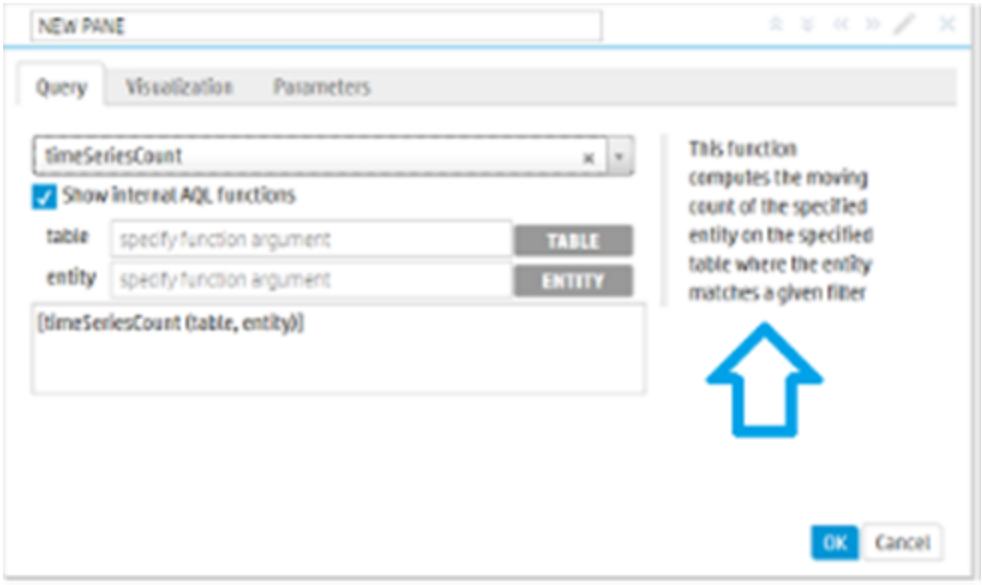
Tip: Use the `bpm_functions.aql` module as an example. This AQL module contains several AQL functions that can be used as a template for creating your own. They reside in the `$OPSA_HOME/inventory/lib/hp/aql` directory.

You can also view these AQL functions when you use the **Add A Query Pane** option from an Operations Analytics dashboard. See *Dashboards and Query Panes* in the *Operations Analytics Help* for more information.

Note: To view the AQL query associated with each AQL function provided by Operations Analytics, look at the `.aql` files in the `$OPSA_HOME/inventory/lib/hp/aql` directory or use the `opsa-aqlmodule- manager.sh` command.

When creating AQL functions to be imported, note the following::

- The comment preceding each AQL function is displayed as the description for the AQL function selected as shown in the following example:



- As a best practice, name your file using an `.aql` extension.
- As a best practice, use the validate option in the `opsa-aql-module-manager.sh` script to ensure your module will import.
- As a best practice, place your file in the `$OPSA_HOME/inventory/lib/user/aql` directory before it is imported. This helps to ensure that the file is not overwritten when upgrading to a new Operations Analytics version.
- To make your AQL functions available to your user community, use the `opsa-aql-modulemanager.sh` script. This script imports the AQL functions defined in your module into the Operations Analytics database and makes them available to your user community by default. See the `opsa-aqlmodule- manager.sh` reference page (or the Linux manpage) for more information.

Creating and Using AQL Functions

When building AQL queries, you can also define AQL functions or expressions. AQL functions are functions that can be used in place of an associated AQL query. AQL functions are a convenient way of defining and naming frequently used AQL queries for reuse. When you define the AQL function, you name the AQL function, define its arguments and the associated AQL query as well as the argument values to pass to that AQL query. You can define your AQL functions using a text editor, then import them in Operations Analytics.

To select an AQL Function provided by Operations Analytics, use the **Add a Query Pane** feature from an Operations Analytics dashboard. See *Dashboards and Query Panes*in the Operations Analytics help for more information.

To create an AQL function use the following syntax:

`define <AQL function name>(argument_1, argument_2,...argument_n)=<AQL query syntax>`

Arguments are those values that are passed to the associated AQL function. Any value that is used in the AQL query is known as a parameter. For example, the name of a host might be a valid parameter for an AQL query.

To use an AQL function use the following syntax:

`[<AQL function name>(value for argument_1, value for argument_2,...value for argument_n)]`

The brackets ([]) are mandatory.

Note the following:

- You create AQL functions using a text editor.
- To make the AQL functions available to your user community, import the AQL functions using the `opsa-aql-import.sh` script.
- The arguments that can be passed to an AQL function include any parameter included in an AQL query.

AQL Function Syntax

Name of the AQL Function	Description	Example
AQL_function_name	Name of the AQL function. **Tip:** Use a name that will help you to remember the AQL function purpose. Alphanumeric characters and underscore (_) are permitted. Spaces and other special characters (~ ! @ # $ % ^ &; * ( ) + -) are not permitted.	`cpu_threshold`
argument_n	The nth argument to be passed to the associated AQL query. You can enter any number of arguments. **Note:** In this example, **percent** is used to identify the cpu utilization percent threshold.	`percent`

AQL Function Syntax, continued

Name of the AQL Function	Description	Example
AQL_query_syntax	Syntax for the AQL query to which the AQL function is associated. See the next table for more detail. **Note:** When the AQL function is used, each argument value provided is passed.to the associated AQL query. See the **bold** text in the example.	from i in (oa_sysperf_global) let analytic_interval=between ($starttime,$endtime) where (aggregate_avg(i.cpu_util > **percent**) group by i.host_name select i.host_name, aggregate_avg (i.cpu_util)

The following table shows more detail for the AQL_query_syntax example shown in the previous table.

Syntax for the AQL_query_syntax Example

`define cpu_threshold(percent) =`	from i in (oa_sysperf_global) let analytic_interval=between($starttime,$endtime) where (aggregate_avg(i.cpu_util) > percent) group by i.host_name select i.host_name, aggregate_avg(i.cpu_util)

To use the `cpu_threshold` AQL function to return a list of all the hosts where the average CPU utilization exceeds 80 percent, include the following parameter values: `[cpu_threshold(0.8)]`

The following AQL function selects the host name that matches the value of argument **name**. The query returns the following information for the most recent **number** of OMi events that originated from the host selected:

- host name (hostinfo_dsname)
- timestamp
- message title
- severity

define `host_events (name,number) =`	from i in (omi_events_omievents) where (i.hostinfo_dnsname like **name**) let analytic_interval = between($starttime,$endtime) let offset = 0 let limit = **number** select i.hostinfo_dnsname, i.timestamp, i.title, i.severity

To use the `host_events` AQL function to return a list of the most recent 50 events for all hosts in the "enterprise.com" domain, include the following argument values:

```
[host_events("enterprise.com", 50)]
```

Importing Analytic Query Language Functions

Use the opsa-aql-module-manager.sh script to manage the AQL functions that you create. When using the opsa-aql-module-manager.sh script, note the following:

- You must specify the tenant name for which the AQL functions should be available.
- Use file names that identify the types of AQL functions contained in each file.
- You define the `<module_name>` in the first line of each file; for example: `module <my_new_module>`;
- You validate, list, and delete modules using the module name.

Use the opsa-aql-module-manager.sh script to perform the following tasks:

Validate the AQL functions included in n module file

Run the following command: `opsa-aql-module-manager.sh -t <tenant_name> -v <file_name>`

Note: The opsa-aql-module-manager.sh script does not currently detect some syntax errors, such as unbound variables referenced within the body of an AQL function. Take extra care when creating and editing your AQL functions.

Import an AQL Module

Enter the following command: `opsa-aql-module-manager.sh -t <tenant_name> -i <file_name>`

When importing AQL functions, note the following:

- After importing your AQL functions, all functions are available to the user community in the specified tenant.
- To replace or redefine AQL functions, you must make the appropriate changes to the .aql module, then re-import the file.

List all AQL modules that have been imported into Operations Analytics

Run the following command: `opsa-aql-module-manager.sh -t <tenant_name> -l modules`

List the AQL functions contained in a module that has been imported into Operations Analyticss

Run the following command: `opsa-aql-module-manager.sh -t <tenant_name> -l <module_name>`

See the *opsa-aql-module-manager.sh* reference page (or the Linux manpage) for more information.

Collection-Specific AQL Functions

Operations Analytics has a notion of a content pack, which is how additional functionality is added to the product. Basically, a content pack consists of the following:

- A collection or collections.
- AQL functions to analyze one or more of the new collections.
- Dashboards to present the analytics for the new collection or collections.

AQL functions that are specific to a particular collection are usually specialized to certain types of metrics and analytics. They provide a very easy way for the user to ad hoc analysis on the data. The following examples show Oracle-specific AQL functions.

/* Returns the top N of an aggregate analytic on an HP Operations Oracle SPI metric. Input parameters are the host filter, database instance filter, metric name, aggregate analytic, and N. */

```
define oaOraperfTopNAggregateMetric
(hostFilter,instanceFilter,metric,aggregate_analytic,N) =
from i in (oa_oraperf_graph)
let analytic_interval = between($starttime, $endtime)
let interval=$interval
let aggregate_playback=$aggregate_playback_flag
where ( ( i.host_name like hostFilter ) && ( i.db_instance_name like instanceFilter
) )
group by i.host_name, i.db_instance_name
select topN(aggregate_analytic(i.metric), N);
```

/* Returns the aggregate analytic above a specified threshold on an HP Operations Oracle SPI metric. Input parameters are the host filter, database instance filter, metric name, aggregate analytic, and threshold percentage. */

```
define oaOraperfAggregateMetricAbovePctile
(hostFilter,instanceFilter,metric,aggregate_analytic,upper_limit_pctile) =from i in
(oa_oraperf_graph)
let analytic_interval = between($starttime, $endtime)
let interval=$interval let aggregate_playback=$aggregate_playback_flag
where ( ( ( i.host_name like hostFilter ) && ( i.db_instance_name like
instanceFilter ) ) && ( aggregate_analytic(i.metric) > inverse_pctile(aggregate_
```

```
analytic(i.metric), upper_limit_pctile ) ) )
group by i.host_name, i.db_instance_name
select aggregate_analytic(i.metric);
```

Generic AQL Functions

Generic functions are more generalized and can be used on any type of collection. There are two primary generic functions:

- `metricQuery`
- `attributeQuery`

The generic functions are mostly a template or shorthand for composing a complete query. These functions are used by PQL in the process of generating the AQL for dashboard panes. Because of their succinctness they are also frequently used in the out-of-box dashboards.

`metricQuery` takes four parameters using the following syntax:

`metricQuery(<table name>, {<where clause>), {<group by>}, {<select>})`

`metricQuery` (as the name suggests) is intended as a generalized approach to formulate a query on metrics that yields either time series metric data or aggregated metric data.

Note: The '{' delimiters are used instead of normal '(' to group the clauses.

An example AQL expression use of `metricQuery` is:

```
[metricQuery(oa_sysperf_global, {(i.host_name ilike "*")}, {i.host_name}, {moving_
avg(i.active_processes), moving_avg(i.cpu_util)}]
```

(that is, select time series data of `active_processes` and `cpu_utilization` for all hosts in the `oa_sysperf_global` collection).

`attributeQuery` takes three parameters using the following syntax:

`metricQuery(<table name>, {<where clause>), {<select>})`

`attributeQuery` is intended as a generalized approach to formulate a query on attributes that yields single or aggregated attribute data.

Note: The '{' delimiters are used instead of normal '(' to group the clauses.

An example AQL expression use of `attributeQuery` is:

```
[attributeQuery(oneview_rest_inventory, {(i.category_name == "enclosures")},
{i.name}]
```

(that is, select the name of all enclosures from the `oneview_rest_inventory` collection).

AQL Expressions

AQL expressions include multiple AQL functions. Use AQL expressions when you want the results of multiple queries to be combined into a single query pane in a dashboard.

You can use AQL functions in an AQL expression in any of the following ways:

Use a single AQL function

Syntax: `[<aql_function_invocation>]`

Concatenate multiple AQL functions

Concatenating multiple AQL functions enables you to concatenate the results from each AQL function as if they were run individually.

Syntax: `[<aql_function1>,<aql_function2>, ….<aql_functionn>]`

The following AQL function returns the concatenation of the results from the following:

- moving averages of CPU utilization
- moving distinct count of host names monitored by the HP Operations Agent

```
[oaSysperfMovingMetric("*.mydomain.com", cpu_util, moving_avg),
oaSysperfHostsMovingCount("*.mydomain.com")]
```

/* Returns the moving aggregation analytic function results for the specified metric. Input parameters are host filter, metric, and analytic function. */

```
define oaSysperfMovingMetric(hostFilter, metric, moving_analytic) =
from i in (oa_sysperf_global)
let analytic_interval = between($starttime, $endtime) let interval = $interval
where i.host_name like hostFilter
group by i.host_name
select moving_analytic(i.metric);
```

/* Returns moving distinct count of hosts being monitored by the HP Operations Agent. Input parameter is the host filter. */

```
define oaSysperfHostsMovingCount(hostFilter) =
from i in (oa_sysperf_global)
let analytic_interval = between($starttime,$endtime)
let interval = $interval where i.host_name like hostFilter
select moving_distinct_count(i.host_name);
```

Use multiple AQL functions so that the results from one AQL function is an input filter for another AQL function

This type of AQL expression is known as an AQL composition.

Syntax: `[do <target_function> filter by <filter_function> with <filter_criteria>]`

`<target_function>` is the AQL function to run.
`<filter_function>` is the AQL function used to filter the results.

<filter_criteria> is the criteria to use for filtering the results of target function. The syntax of <filter_criteria> is:
(<filter_criteria_element1>, <filter_criteria_element2>, ….)

Each <filter_criteria_element> specifies a metric or attribute column name with its associated collection. Values for the column name specified must be returned in the target_function and filter_Function results.

Note: All of the filter criteria elements must be met to successfully filter the target function results.

The syntax for any filter criteria element is:

<target_function_name>.<target_function_resultcolumn> == <filter_function_name>.<filter_function_resultcolumn>

The <target_function_resultcolumn> can be any of the expected result columns from the results of <target_function>.

<target_function_name> is the name of the target function.

Similarly, <filter_function_resultcolumn> can be any of the expected result columns from the results of <filter_function>. The <filter_function_name> is the name of the filter function.

The following example AQL expression returns the moving_avg, moving_max, and moving min of CPU utilization for the top five hosts with the highest aggregate_avg cpu_util values.

[do oaSysperfMovingMetricAvgMaxMin("*", cpu_util) filter by oaSysperfTopNAggregateMetric ("*.mydomain.com",cpu_util,aggregate_avg,5) with (oaSysperfMovingMetricAvgMaxMin.host_name== oaSysperfTopNAggregateMetric.host_name)]

/* Returns the moving average, maximum, and minimum values of a specific metric by host. Input parameters are the host filter and the metric. */

```
define oaSysperfMovingMetricAvgMaxMin(hostFilter, metric) =
from i in (oa_sysperf_global)
let analytic_interval = between($starttime,$endtime) let interval = $interval
where i.host_name like hostFilter
group by i.host_name
select moving_avg(i.metric), moving_max(i.metric), moving_min(i.metric);
```

/* Returns the topN of a moving aggregate analytic function on a metric. Input parameters are the host filter, metric, moving aggregate analytic function, and N. */

```
define oaSysperfTopNMovingMetric(hostFilter, metric, moving_analytic, N) =
from i in (oa_sysperf_global)
let analytic_interval = between($starttime, $endtime) let interval = $interval
where i.host_name like hostFilter group by i.host_name
select topN(moving_analytic(i.metric), N);
```

The following AQL expression returns the aggregate_avg CPU utilization for all server nodes in the Operations Analytics topology. These servers include the database server nodes. This example uses topology data to filter and return metric analysis for important entities in your topology:

```
[do oaSysperfAggregateMetric("*",cpu_util,aggregate_avg) filter by opsaNodes()
with (
oaSysperfAggregateMetric.host_name== opsaNodes.opsa_server_name,
```

```
oaSysperfAggregateMetric.host_name== opsaNodes.collector_server_name,
oaSysperfAggregateMetric.host_name== opsaNodes.logger_server_name,
oaSysperfAggregateMetric.host_name== opsaNodes.vertica_node
)]
```

/* Returns the results of the overall aggregate analytic function applied to the specified metric. Input parameters are host filter, metric, and overall aggregate analytic function. */

```
define oaSysperfAggregateMetric(hostFilter,metric,aggregate_analytic) =
from i in (oa_sysperf_global) let analytic_interval = between($starttime, $endtime)
where i.host_name like hostFilter
group by i.host_name
select aggregate_analytic(i.metric);
```

/* Returns the host names of Operations Analytics application servers, logger servers, collector servers, and vertica nodes in an Operations Analytics deployment */

```
define opsaNodes() = from i in (opsa_topology) select i.opsa_server_name, i.logger_
server_name, i.collector_server_name, i.vertica_node;
```

/*Compares SiteScope response times with OA performance metrics*/

```
[do metricQuery({oa_sysperf_global}, {1==1}, {i.host_name,i.cpu_util}, {moving_avg
(i.cpu_util), moving_avg(i.disk_byte_rate)})
filter by topoQuery(custom_topology_nodegroup,{(i.service_name ilike "*")},
{i.service_name,i.group_name,i.host_name})
with (metricQuery.host_name == topoQuery.host_name ), do metricQuery({sitescope_
urlmonitor_metrics}, {1==1},{i.target_name},{moving_avg(i.roundtrip_time_
milliseconds)})
filter by topoQuery(custom_topology_nodegroup,{(i.service_name ilike "*")},
{i.service_name,i.group_name,i.host_name})
with (metricQuery.target_name == topoQuery.host_name ), do metricQuery({sitescope_
ping_metrics}, {1==1},{i.target_name},{i.target_name,moving_avg(i.round_trip_
time)})
filter by topoQuery(custom_topology_nodegroup,{(i.service_name ilike "*")},
{i.service_name,i.group_name,i.host_name})
with (metricQuery.target_name == topoQuery.host_name )]
```

Generic AQL Outer Functions

Bucket Function

The `bucket` function is used to group the counts of items in a data set that fall into partitions. An example use scenario is that given the overall cpu_utilization of a set of hosts, you would like to see how many fall into the 0-10 percent, 10-20 percent, and so forth.

The parameters to the bucket function are:

- `AQL expression` – This is the aql query that yields the data set to partition.
- `numbuckets` (optional) – This is an integer to define the number of partitions.
- `min` and `max` (optional) – These two numbers specify the range of values to partition.
- `aliasforbucketmemberscount` (optional) – This is a label to provide a meaningful name for the count of items in each partition.

Consider the following example use case of the bucket function:

```
[bucket[metricQuery(oneview_rabbitmq_metrics,{i.category=="server-hardware"},
{i.resource_uri},{aggregate_avg(i.cpu_utilization)})]
(numbuckets=5,min=0,max=100,aliasforbucketmemberscount="Number Of Servers")]
```

- `numbuckets`: Change the value of this parameter to the number of partitions you want to display. The default value is 5 if you do not assign a value.
- `min`: Change the value of this optional parameter to the minimum data value you want partitioned. min is an optional parameter. If you use it, you must use it along with the `max` parameter for this parameter to function correctly.
- `max`: Change the value of this optional parameter to the maximum data value you want partitioned. max is an optional parameter. If you use it, you must use it along with the `min` parameter for this parameter to function correctly.

  Note: If you do not specify the `min` and `max` parameters, the range (`min`/`max`) is automatically calculated and the entire range of values are partitioned into buckets.

- `Aliasforbucketmemberscount`: Change the value of this optional parameter if you need a meaningful name for the count of items in each partition. If this parameter is not specified, then the "bucketmemberscount" string is used as the label.

Chapter 5: Arithmetic Expressions and Aliases

Using Arithmetic Expressions and Aliases in AQL

Operations Analytics collections contain multiple attribute and multiple metric columns. When creating Operations Analytics dashboards, it is helpful to combine multiple metric columns using arithmetic expressions and query the expression instead of the individual metric columns. It is also helpful to combine columns and another constant in an arithmetic expression, then query the expression instead of the individual metric columns. It is also helpful to combine columns and another constant in an arithmetic expression, then query the expression. It is also desirable to invoke statistical intrinsic functions such as `moving_avg`. `moving_max`, `moving_min`, `moving_total`, `aggregate_avg`, `aggregate_max`, `aggregate_min`, `aggregate_total`, `pctile`, `topN`, `rank`, and others on such arithmetic expressions.

Finally, a naming or aliasing facility must make it convenient to use arithmetic expressions. You should be able to specify a custom name or alias for your arbitrary arithmetic expressions.

In this chapter we will discuss how AQL enables you to use arithmetic expressions and aliases.

Arithmetic Expressions and Alias Support in Language Core

AQL supports the following arithmetic operators: unary negative (unary -), addition (+), subtraction (-), multiplication (*), and division (/). You can construct arbitrarily complex arithmetic expressions using these operators.

You can also use higher order AQL intrinsic analytic functions such as `moving_avg`. `moving_max`, `moving_min`, `moving_total`, `aggregate_avg`, `aggregate_max`, `aggregate_min`, `aggregate_total`, `pctile`, `topN`, `rank`, and others on such arithmetic expressions.

AQL permits you to specify aliases using the 'as' keyword. This is useful for providing meaningful names to arithmetic expressions. Aliases used in AQL will influence the resulting Operations Analytics console labels on dashboards.

You can also use aliases with any measurement or dimension expression in AQL statements regardless of whether an arithmetic expression is used or not.

Arithmetic expressions and aliases are permitted in `select` clause selectors and filtering relational expressions involving predicates in an AQL statement's `where` clause. You can also use aliases with dimension expressions used in an AQL statement's `group by` clause.

The following is an example of a raw AQL statement to query a raw metric arithmetic expression:

```
from i in (foo_collection) let analytic_interval=between($starttime,$endtime)
select i.host_name, i.cpu_util_fraction*100 as "cpu util in %"
```

The above example illustrates the use of an arithmetic expression involving a * operator, a constant (100), and the `column cpu_util_fraction` metric to come up with a more meaningful cpu utilization in the percentage column at query time.

The following example demonstrates the use of an arithmetic expression involving two metric columns, `cpu_util` and `mem_util` along with the subsequent invocation of the `moving_max` intrinsic on the resulting expression.

```
from i in (oa_sysperf_global)
let analytic_interval=between($starttime,$endtime) let interval=$interval
group by i.host_name
select moving_max(i.cpu_util*i.mem_util/100 as "foo coeff")
```

The following example shows the use of an arithmetic expression as a filtering condition in the `where` clause predicate.

```
from i in oa_sysperf_global
let analytic_interval=between($starttime,$endtime) let interval=$interval
where i.host_name == "foo.bar.com" && moving_avg(i.cpu_util*i.mem_util/100 as "foo
coeff") >= 1.0
group by i.host_name
select moving_avg(i.cpu_util*i.mem_util/100 as "foo coeff")
```

The following example illustrates the use of a higher order intrinsic `topN` on an `aggregate_avg` of an arithmetic expression. Also note the use of the "foo hosts" alias to give a custom name to a group by a dimension or attribute host_name.

```
from i in oa_sysperf_global
let analytic_interval=between($starttime,$endtime) let interval=$interval
let aggregate_playback=$aggregate_playback_flag
group by i.host_name as "my hostname column"
select topN(aggregate_avg(i.cpu_util*i.mem_util/100 as "foo coeff"))
```

The following example demonstrates the use of the ("host count") alias for a non- arithmetic expression measurement selected in an AQL:

```
from i in oa_sysperf_global
let analytic_interval=between($starttime,$endtime) let interval=$interval
group by i.source
select moving_distinct_count(i.host_name) as "host count"
```

Arithmetic Expressions and Aliases in AQL Functions

Arithmetic expressions and aliases are also allowed in parameters to AQL functions. Such AQL function parameters include those representing parts of `select` clauses and `where` clauses. Additionally, aliases can be specified for group by attributes or dimensions in parameters representing parts of `group by` clause.

AQL functions support arithmetic expressions and aliases in the following two ways:

- Aliased arithmetic expressions could be parts of parameters enclosed in {} function parameter start and end markers as shown in the **bold text** in the following example:

  ```
  [metricQuery(oneview_rabbitmq_metrics,{i.resource_category=="server-hardware"},
  {i.resource_uri},{moving_avg(i.average_power/i.power_capacity as "power
  coefficient")}]
  ```

In the above example, you pass the full arithmetic expression and its alias as a function parameter enclosed within the {} markers for AQL function parameters.

Similarly, you can use arithmetic expressions as operands to influence the `where` clause filter conditions or predicates as shown in the bold text in the following example:

```
[metricQuery(oneview_rabbitmq_metrics,{(i.resource_category=="server-hardware")
&& (aggregate_avg(i.average_power/i.power_capacity as "power coefficient" >
0.5))}, {i.resource_uri},{aggregate_avg(i.average_power/i.power_capacity as
"power coefficient")}]
```

In the above example, you set up a filter to fetch only those average power coefficients that are greater than 0.5.

The following is an example of invoking an AQL function to compute the `topN` of `aggregate_avg` of a metric arithmetic expression:

```
[metricQuery(oa_sysperf_global,{},{i.host_name},{topN(agreggate_avg(i.cpu_
util*i.mem_util/100 as "foo coeff"))})]
```

The following is an example of using an arithmetic expression as a filtering condition in a `where` clause predicate:

```
[metricQuery(oa_sysperf_global,
{i.host_name == "foo.bar.com" && moving_avg(i.cpu_util*i.mem_util/100 as "foo
coeff") >= 1.0}, {i.host_name}, {moving_avg(i.cpu_util*i.mem_util/100 as "foo
coeff"))})]
```

Although the above examples are using the generic metricQuery() AQL function, similar approaches apply to user defined custom AQL functions.

- Parts of aliased arithmetic expressions could be passed as AQL function parameters. For example, you can create a custom function that takes two metric columns of a collection, then computes an `aggregate_avg` of the product of the columns. The definition could look like the following:

  ```
  define
  getProductAverages(metric1,metric2,aliasforhost,aliasforproduct) =
  from i in (foo_collection)
  let analytic_interval=between($starttime,$endtime) let interval=$interval
  let aggregate_playback=$aggregate_playback_flag
  group by i.host_name as aliasforhost
  select aggregate_avg(i.metric1*i.metric2 as aliasforproduct)
  ```

  An example invocation for the above custom function could look like the following:

  ```
  [getProductAverages(cpu_util, mem_util, myhost, myproduct)]
  ```

Arithmetic Expressions and Aliases in AQL Concatenations and `do` filter by expressions

You can construct AQL concatenation or AQL do filter by expressions by using AQL functions with parameters having aliased arithmetic expressions or by the simple use of aliases on non-arithmetic expressions.

The following is an example of AQL concatenation with arithmetic expressions and the simple use of an alias on non-arithmetic expressions:

```
[metricQuery(oa_sysperf_global, {i.source like "HP*"}, {i.source}, {moving_
distinct_count(i.host_name) as "HP* host count"}),
metricQuery(oa_sysperf_global, {i.source like "HPACollector*"}, {i.source},
{moving_distinct_count(i.host_name) as "HPACollector* host count"})]
```

As another example, suppose you create the following AQL and expect to see two lines displayed in the resulting dashboard, but Operations Analytics displays only one line:

```
[metricQuery({bpm_application_performance}, {i.transaction_response_time >50 &&
i.application ilike "ART*"}, {i.application}, {i.transaction_response_time}),
metricQuery({bpm_application_performance}, {i.transaction_response_time <50 &&
i.application ilike "ART*"}, {i.application}, {i.transaction_response_time})]
```

To resolve this issue, use aliases as shown in the following AQL to obtain the two lines you expected:

```
[metricQuery({bpm_application_performance}, {i.transaction_response_time >50 &&
i.application ilike "ART*"}, {i.application}, {i.transaction_response_time as
"above 50"}),
metricQuery({bpm_application_performance}, {i.transaction_response_time <50 &&
i.application ilike "ART*"}, {i.application}, {i.transaction_response_time as
"below 50"})]
```

The following is an example of an AQL do filter by expression with arithmetic expressions in a do function. This example assumes that you have used the Operations Analytics topology manager to create a node group service topology named service1:

```
[do metricQuery(oa_sysperf_global,{},{i.host_name },{moving_avg(i.cpu_util*i.mem_
util/100 as "foo coeff")}) filter by topoQuery(custom_topology_nodegroup,
{i.service_name == "service1"},{i.service_name,i.host_name}) with
(metricQuery.host_name==topoQuery.host_name)]
```

You can create an AQL do filter by expression with aliased dimensions used in a with clause of a do filter by expressions as shown in the following examples:

Example 1:

```
[do metricQuery(oa_sysperf_global,{},{i.host_name as "bar host" },{moving_avg
(i.cpu_util)})
filter by
topoQuery(custom_topology_nodegroup,{i.service_name == "mynodegroupservice"},
{i.service_name,i.host_name as "foo host"})
with (metricQuery."bar host" == topoQuery."foo host") ]
```

Note the use of aliases in the do function and the filter function, and the corresponding use of aliases in the with clause.

Example 2:

```
[do metricQuery(oa_sysperf_global,{},{i.host_name},{moving_avg(i.cpu_util)})
filter by
topoQuery(custom_topology_nodegroup,{i.service_name == "mynodegroupservice"},
{i.service_name,i.host_name as "foo host"})
with (metricQuery.host_name==topoQuery."foo host") ]
```

Note the use of the alias in the filter function alone and the corresponding `with` clause specification in terms of only the alias in the filter function.

In general, if you specify an alias for an attribute or dimensions in either a do function or a filter function, then use the same attributes in a `with` clause, make sure you use the corresponding aliases to develop the `with` clause.

More about Alias Support and Alias Placement Conventions

AQL supports the following character set in aliases:

- Alphanumeric characters.
- Special characters: space, _ (underscore), - (hyphen), %, #, $,!.
- Using surrounding double quotes ("") for aliases containing embedded spaces.

Note: AQL supports a maximum of 128 characters in an alias.

Alias Placement Conventions for Arithmetic Expressions

You could either place the alias at the innermost arithmetic expression, the outermost measurement expression, or anywhere in between.

Here are a few examples to illustrate these alias placement conventions for arithmetic expressions.

The following AQL example queries the top 20 hosts demonstrating the highest cpu utilization percentage aggregate averages. Notice the specification of alias at the inner most expression level shown in bold font.

```
from i in (foo_collection) let analytic_interval=between($starttime,$endtime) let
interval=$interval
let aggregate_playback=$aggregate_playback_flag
where (i.host_name like "*")
group by i.host_name
select topN(aggregate_avg(i.cpu_util_fraction*100 as "cpu util percent"), 20)
```

In the above example, the Operations Analytics console displays **cpu util percent (Aggregate Avg)** as the label for these measurements.

You can rewrite this example as follows for the Operations Analytics console to display **cpu util percent agg avg** as the label for the measurement, (note the specification of alias at the inner aggregate_avg intrinsic invocation level):

```
from i in (foo_collection) let analytic_interval=between($starttime,$endtime) let
interval=$interval
let aggregate_playback=$aggregate_playback_flag
where (i.host_name like "*")
group by i.host_name
select topN(aggregate_avg(i.cpu_util_fraction*100) as "cpu util percent agg avg",
20)
```

You can specify the alias at the outermost level as shown in the following example:

```
from i in (foo_collection) let analytic_interval=between($starttime,$endtime) let
interval=$interval
let aggregate_playback=$aggregate_playback_flag
where (i.host_name like "*")
group by i.host_name
select topN(aggregate_avg(i.cpu_util_fraction*100), 20) as "topn cpu util percent
agg avg"
```

For consistency with how Operations Analytics treats cases of non-arithmetic expressions without aliases specified, it is recommended that you use aliases for arithmetic expressions at the innermost level to identify the metric arithmetic expression.

Higher Order Arithmetic Involving Intrinsic Calls

So far you have seen how you can combine multiple columns and constants to compose arithmetic expressions and query either these raw expressions or intrinsic calls on those expressions. This information included examples of arithmetic expression usage with the following intrinsic functions: `moving_avg`, `moving_max`, `moving_min`, `moving_total`, `aggregate_avg`, `aggregate_max`, `aggregate_min`, `aggregate_total`, `pctile`, `topN`, `rank`, and others.

In this section, you will look at a capability in AQL that enables you to initiate higher order arithmetic expressions among intrinsic calls and constants. This capability, when exercised, enables you to fulfill vector arithmetic use cases if you consider the results of individual intrinsic calls as some kind of result vectors. This is especially relevant for cases where the intrinsic call is such that a vector of values at multiple timestamps (time series) is returned as results. For example, you might first want to calculate the `moving_total` of failed calls, then calculate a `moving_total` of all calls, and finally calculate a timeseries that represents the fraction of all calls that failed at different times. For these use cases, it is possible to use the intrinsic calls in higher order arithmetic expressions involving one or more intrinsic calls, constants, or both.

Just as AQL supports arithmetic operators in the basic collection column, constant arithmetic expressions,or both, AQL supports the following arithmetic operators in higher order arithmetic expressions too: unary negative (unary -), addition (+), subtraction (-), multiplication (*), and division (/). You can construct arbitrarily complex arithmetic expressions using these operators and one or more intrinsic calls or constants.

Similarly AQL permits you to specify aliases using the 'as' keyword for such higher order arithmetic expressions thus providing meaningful names to these higher order arithmetic expressions.

These higher order arithmetic expressions and their aliases are permitted in `select` clause selectors and filtering relational expressions involving predicates in an AQL statement's `where` clause.

The following example shows a raw AQL statement that queries the higher order arithmetic expression:

```
from i in (foo_call_collection) let interval = $interval
let analytic_interval=between($starttime,$endtime)
select i.region,
topN( (moving_total(i.num_calls) - moving_total(i.num_completed_calls)) / moving_
total(i.num_calls) * 100 as "Failed Call %" )
```

The example shown above illustrates the use of a higher order arithmetic expression that accomplishes the failed call percentage calculation described in the first few paragraphs of this section. It computes the following:

- The moving total intrinsic of all calls (the `num_calls` column) for each distinct region attribute value in the foo_call_collection.
- The moving total of completed calls (the `num_completed_calls` column) for each distinct region.
- Evaluates a vector coefficient of this difference vector and the moving total of the `num_calls` vector for each distinct region.
- Multiplies the resulting coefficient by 100 to obtain the failed call % vector showing a trend of failed calls % over time for each distinct region attribute value in the foo_call_collection.
- Performs a `topN` intrinsic for this % vector expression to get the top intervals for each region.

You can visualize the result of the computation done in this example on an Operations Analytics visualization, such as a bar chart, to understand the top intervals with the highest failed call percentages.

You can also use basic arithmetic expressions involving multiple columns of a collection as arguments to any intrinsic call that is, in turn, involved in a higher order arithmetic expression involving other intrinsic calls. The following example illustrates this aspect of these higher order expressions:

```
from i in (foo_call_collection) let interval = $interval
let analytic_interval=between($starttime,$endtime)
select i.region,
topN( (moving_total(i.num_calls-i.num_calls_to_ignore as effective_num_calls) -
moving_total(i.num_completed_calls)) / moving_total(i.num_calls-i.num_calls_to_
ignore as effective_num_calls) * 100 as "Failed Call %" )
```

In the above example, it first deducts a hypothetical column that provides a to-be-ignored call count from the overall total calls, then evaluates the moving total intrinsic on the difference before feeding that difference to the rest of the higher order arithmetic expression for the Failed Call %.

Such higher order arithmetic expressions are also permitted in parameters supplied to AQL functions. Such AQL function parameters include those representing parts of `select` clauses and `where` clauses.

The following example illustrates invoking the same failed call % example by using an AQL function:

```
[metricQuery(foo_call_collection,{},{},{region, topN((moving_total(i.num_calls) -
moving_total(i.num_completed_calls)) / moving_total(i.num_calls) * 100 as "Failed
Call %")})]
```

Similarly, you can construct an AQL `do` filter by using expressions or AQL concatenation expressions by using AQL functions with parameters having aliased higher order arithmetic expressions.

The following is an example of an AQL `do` filter by expression with higher order arithmetic expressions in a `do` function. This example assumes that you have used the Operations Analytics topology manager to create a node group service topology named `service1`:

```
[do metricQuery(oa_sysperf_global,{},{i.host_name },{moving_avg(i.cpu_util)
/aggregate_distinct_count(i.sourceid) * 100 as "foo coeff")}) filter by topoQuery
(custom_topology_nodegroup,{i.service_name == "service1"},{i.service_name,i.host_
name}) with (metricQuery.host_name==topoQuery.host_name)]
```

This example attempts to calculate the resultant time series vector that is a result of evaluating the coefficient of cpu utilization moving average time series vector and flat aggregate count (a scalar) of distinct sources per host for all hosts defined to be part of service `service1`.

It is also possible to come up with a higher order arithmetic expression involving only flat aggregate (scalars) to accomplish a complex scalar arithmetic use case. For instance, the following example permits us to get the coefficient of aggregate total of `num_calls` for each country divided by the total count of regions for each country:

```
[metricQuery(foo_call_collection,{},{},{country, aggregate_total(i.num_calls)
/aggregate_distinct_count(i.region) as "My Distribution Coefficient"})]
```

Note: Just like in the case of using basic arithmetic expressions described in previous sections, the columns used in the higher order arithmetic expressions involving intrinsic calls are also expected to be from same collection.

Chapter 6: Analytics Query Language for Log Data

The information in this section explains how to use AQL functions to search log file information. Examples in this topic use AQL to return the information collected by log files configured using HP ArcSight Logger.

Note: The queries in this section do not apply to structured log files. Structured log files are fragments of log file data that are stored as collections in Operations Analytics. Structured logs are log files that are configured as collections. These collections are created so that users can perform analytics on the log file contents.

You can use three types of AQL functions to search log file information:

- To search for text strings, use the `aqlrawlog` function.
- To count the number of log file entries, use the `aqlrawlogcount` function.
- To enter a query supported by HP ArcSight Logger, use the `aqlrawlogarbitrary` function.

Using the aqlrawlog Function to Search for Text Strings

Use the `aqlrawlog` function to search the log file entries stored in HP ArcSight Logger servers.

The `aqlrawlog` query returns the following attributes for each matching log file message entry: `timestamp`, `message text`, `host name`, and `source host name`.

Syntax: `aqlrawlog(<aqllit><text_to_search></aqllit>, <starttime_as_seconds_since_epoch>, <end_time_as_seconds_since_epoch>, ""|"<comma_separated_list_of_logger_host_names>"[,<limit>])`

Note: When using Splunk as the log data source, do not use the keyword `"host="` in the search string. For example, using a search string that includes `aqlrawlog(<aqllit>host ="10.60.15*"</aqllit>, $starttime, $endtime, "", $limit)` results in no matches. Instead, use a search string like `aqlrawlog(<aqllit>"10.60.15*"</aqllit>, $starttime, $endtime, "", $limit)` to obtain matches.

aqlrawlog arguments

`[let timeout=<timeout_in_seconds>]`
`[let limit=<limit>]`
`<text_to_search>` is the text string that must match in the log file entries.

Note: The `<text_to_search>` argument must be enclosed by the `<aqllit>` keyword. For example `<aqllit>severity</aqllit>`.

<*starttime_as_seconds_since_epoch*> is the start time of the time window within which to look for matching log file entries.

Note: To use the value selected in the Operations Analytics console, enter `$starttime` as the value for this argument.

<*endtime_as_seconds_since_epoch*> is the end time of the time window within which to look for log file entries.

Note: To use the value selected in the Operations Analytics console, enter `$endtime` as the value for this argument.

<*comma_separated_list_of_logger_host_names*> is a comma separated list of host names that identify the HP ArcSight Logger servers to query.

Tip: To query all of theHP ArcSight Logger servers configured for the current tenant, specify `""` as this parameter value.

<limit> is an optional parameter that overrides the default maximum number of log file entries to return.

Note: If you do not use this parameter or the optional `let limit=<limit> clause`, Operations Analytics returns up to a maximum of 2000 log file messages matching the search text. You can also specify `$limit` for this value.

<*timeout_in_seconds*> is the timeout for the search operation. This parameter is specified when using the optional `let timeout=…` clause.

Note: If you do not specify this parameter, Operations Analytics uses the default timeout value.

Examples:

```
/* Returns a maximum of 500 log file entries that include "error" */
aqlrawlog(<aqllit>error</aqllit>, $starttime, $endtime, "", 500)

/*Returns the default maximum number of log file entries that include "error". This
query searches log file entries only on the following servers:
mylogger1.mydomain.com and mylogger2.mydomain.com logger servers*/
aqlrawlog(<aqllit>error</aqllit>, $starttime, $endtime,
"mylogger1.mydomain.com,mylogger2.mydomain.com")

/* Returns the default maximum number of log file entries that include "error". It
uses the timeout value of 5 minutes */
aqlrawlog(<aqllit>error</aqllit>, $starttime, $endtime, "") let timeout=300

/* Returns a maximum number of 500 log file entries that include "error". It uses
the timeout value of 5 minutes */
```

```
aqlrawlog(<aqllit>error</aqllit>, $starttime, $endtime, "") let timeout=300 let
limit=500
```

Use the aqlrawlogcount Function to Count the Number of Log File Entries

Use the `aqlrawlogcount` function to count the log file entries stored in HP ArcSight Logger servers that contain the search text string.

Syntax: `aqlrawlogcount(<aqllit><text_to_search></aqllit>, <starttime_as_seconds_since_epoch>, <end_time_as_seconds_since_epoch>, "" | "<comma_separated_list_of_logger_host_names>","" | "<comma_separated_list_of_group_by_fields>" [,<granularity_in_seconds>])`

Description of each of the aqlrawlogcount arguments

`[let timeout=<timeout_in_seconds>]`

`[let limit=<limit>]`

`<text_to_search>` is the text string that must match in each log file entry returned.

Note: The `<text_to_search>` argument must be enclosed by the `<aqllit>` keyword, for example `<aqllit>severity</aqllit>`.

`<starttime_as_seconds_since_epoch>` is the start time of the time window within which to look for matching log file entries.

Note: To use the value selected in the Operations Analytics console, enter `$starttime` as the value for this argument.

`<endtime_as_seconds_since_epoch>` is the end time of the time window within which to look for matching log file entries.

Note: To use the value selected in the Operations Analytics console, enter `$endtime` as the value for this argument.

`<comma_separated_list_of_logger_host_names>` is a comma separated list of host names that identify the HP ArcSight Logger servers to query.

Tip: To query all of the HP ArcSight Logger servers configured for the current tenant, specify `""` as this parameter value.

`<limit>` is an optional parameter that overrides the default maximum number of log file entries to return.

Note: If you do not use this parameter or the optional `let limit=<limit>` clause, Operations Analytics returns up to a maximum of 2000 log file messages matching the search text. You can also specify `$limit` as the value.

<*timeout_in_seconds*> is the timeout for the search operation specified using the optional `let timeout=…` clause.

Note: If you do not specify this parameter, Operations Analytics uses the default timeout value.

The aqlrawlog query returns the following attributes for each matching log file entry: `timestamp`, `message text`, `host name`, and `source host name`.

<*comma_separated_list_of_group_by_fields*> is a comma separated list of theHP ArcSight Logger attributes in which to group the results.

Tip: If you do not want Operations Analytics to group the results, specify "" as the parameter value.

Note: If you specify "" as this parameter and do not specify <*granularity_in_seconds*>, Operations Analytics computes the moving counts without any group by criteria.

The window of time between <*starttime_as_seconds_since_epoch*> and <*endtime_as_seconds_since_epoch*> is divided into multiple intervals. Operations Analytics calculates counts at each of these intervals. Operations Analytics automatically computes the optimal length of time for each interval.

<*time_interval_in_seconds*> specifies the value Operations Analytics should use to subdivide the window of time between <*starttime_as_seconds_since_epoch*> and <*endtime_as_seconds_since_epoch*>.Operations Analytics computes the moving counts at each of these intervals.

Note: To use the value selected in the Operations Analytics console, enter `$interval` as the value for this argument. See *Dashboards and Query Panes* in the *Operations Analytics Help* for more information about how to specify the `$interval` parameter value in the Operations Analytics console.

<limit> is an optional parameter that overrides the default maximum number of log file entries to return.

Note: If you do not use this parameter or the optional `let limit=`<*limit*> clause, Operations Analytics returns up to a maximum of 2000 log file messages matching the search text. You can also specify `$limit` as the value.

<*timeout_in_seconds*> is the timeout for the search operation specified using the optional `let timeout=…` clause.

Note: If you do not specify this parameter, Operations Analytics uses the default timeout value.

Examples

```
/* Returns the time series counts of log file entries that contain "error" at 5
minuteintervals*/
aqlrawlogcount(<aqllit>error</aqllit>, $starttime, $endtime, "", "", 300)
```

```
/*Returns the time series counts of log file entries that contain "error" for each
combination of deviceHostName and agentSeverity at 5 minute intervals. The function
queries only the mylogger1.mydomain.com server*/
aqlrawlogcount(<aqllit>error</aqllit>, $starttime, $endtime,
"mylogger1.mydomain.com", deviceHostName,agentSeverity, 300)

/*Returns overall aggregate counts of log file entries that contain "error" for
each combination of deviceHostName and agentSeverity. This AQL function queries
only themylogger1.mydomain.com server*/
aqlrawlogcount(<aqllit>error</aqllit>, $starttime, $endtime,
"mylogger1.mydomain.com", "deviceHostName,agentSeverity")

/*Returns the time series of counts of log file entries that contain "error" for
each combination of deviceHostName and agentSeverity at 5 minute intervals. This
AQL function queries only mylogger1.mydomain.com, uses the timeout value of 10
minutes, and queries a maximum of 1000 entries */
aqlrawlogcount(<aqllit>error</aqllit>, $starttime, $endtime,
"mylogger1.mydomain.com", "deviceHostName,agentSeverity", 300) let timeout=600 let
limit=1000
```

Using the aqlrawlogarbitrary Function to Enter a Query Supported by HP ArcSight Logger

Note: A supported query is any query that is configured for use on an HP ArcSight Logger server.

Use the `aqlrawlogarbitrary` function to run any other query supported by your HP ArcSight Logger server.

Operations Analytics displays the `aqlrawlogarbitrary` results table format.

Syntax: `aqlrawlogarbitrary(<aqllit><query_string></aqllit>, <starttime_as_seconds_since_epoch>, <end_time_as_seconds_since_epoch>, ""|"<comma_separated_list_of_logger_host_names>" [,<limit>])`

Description of each of the aqlrawlogarbitrary function arguments.

`[let timeout=<timeout_in_seconds>]`

`[let limit=<limit>]`

`<query_string>` is the query string that is supported by your HP ArcSight Logger server.

Note: The `<query_string>` argument must be enclosed by the `<aqllit>` keyword, for example: `<aqllit>severity</aqllit>`.

`<starttime_as_seconds_since_epoch>` is the start time of the time window within which to look for matching log file entries.

Note: To use the value selected in the Operations Analytics console, enter `$starttime` as the value for this argument.

`<endtime_as_seconds_since_epoch>` is the end time of the time window within which to look for matching log file entries.

Note: To use the value selected in the Operations Analytics console, enter $`endtime` as the value for this argument.

<*comma_separated_list_of_logger_host_names*> is a comma separated list of host names of the HP ArcSight Logger servers to query.

Tip: To query all of the HP ArcSight Logger servers configured for the current tenant, specify `""` as this parameter value.

<`limit`> is an optional parameter and, if specified, it overrides the default maximum rows of information returned by Logger to consider for returning back to the Operations Analytics console.

Note: If you do not use this parameter or the optional `let limit=`<`limit`> clause, Operations Analytics returns up to a maximum of 2000 log file messages matching the search text. You can also specify `$limit` for this value.

<*timeout_in_seconds*> is the timeout for the search operation specified using the optional `let timeout=…` clause.

Note: If you do not specify this parameter, Operations Analytics uses the default timeout value.

Examples

```
/* Returns a maximum of 500 log file entries that contain the text string "error"
*/

aqlrawlogarbitrary(<aqllit>error</aqllit>, $starttime, $endtime, "", 500)

/*Returns up to the default maximum number of log file entries that contain the
text string "error". This AQL function queries only the mylogger1.mydomain.com and
mylogger2.mydomain.com logger servers*/

aqlrawlogarbitrary(<aqllit>error</aqllit>, $starttime, $endtime,
"mylogger1.mydomain.com,mylogger2.mydomain.com")

/* Returns the default maximum number of log file entries that contain "error".
This AQL function uses a timeout value of 5 minutes */

aqlrawlogarbitrary(<aqllit>error</aqllit>, $starttime, $endtime, "") let
timeout=300

/* Returns a maximum of 500 log file entries that contain "error". This AQL
function uses a timeout value of 5 minutes */

aqlrawlogarbitrary(<aqllit>error</aqllit>, $starttime, $endtime, "") let
timeout=300 let limit=500
```

You can add a `let` clause to your `aqlrawlog`, `aqlrawlogcount`, or `aqlrawlogarbitrary` query to define a variable that contains a list of entities returned from an AQL function. The variable can then be used in the `<aqllit>`<*text_to_search*>`</aqllit>` string. This feature is useful when you want to

search for a set of entities, such as hosts, applications, Business Service Management transactions, or database instances without needing to enter the entire list of values.

- To add a let clause to your `aqlrawlog`, `aqlrawlogcount` or `aqlrawlogarbitrary` query, use the following syntax:
  `let <variable_name>=<AQL_function>`

  For example, you could define the `$myhosts` variable to contain the list of servers returned from the AQL function named `oaSysperfHosts`. The `oaSysperfHosts` AQL function uses the following arguments to return hosts that have performance metrics collected:
  `oaSysperfHosts (hostFilter, numHostsLimit)`

  To define a variable to store the results returned from the `oaSysperfHosts` AQL function, use the following syntax:
  `let <variable_name>=oaSysperfHosts (hostFilter, numHostsLimit)`

  For example, to pass the first 50 hosts that have performance metrics collected in the `enterprise.com` domain to the `myhosts` variable, add the following `let` clause to your `aqlrawlog`, `aqlrawlogcount`, or `aqlrawlogarbitrary` query:
  `let myhosts=oaSysperfHosts ("*enterprise.com", 50)`

- The variable you define using the `let` clause can be used in a text search or with a Common Event Field (CEF) field that was configured using the Operations Analytics Log File Connector for MadCap Software ArcSight Logger. See *Configuring the Operations Analytics Log File Connector for ArcSight Logger* in the *HP Operations Analytics Configuration Guide* for more information.

- To use the variable in a text search, use the following syntax: `<aqllit><$variable></aqllit>`
  For example:`<aqllit><$myhosts></aqllit>`

- To use the variable with a CEF, use the following syntax in place of `<aqllit><$variable></aqllit>`:
  `<aqllit><CEF> in [$<variable_name>]</aqllit>`
  For example: `sourcehostName in [$myhosts]`

The previous example searches for all log file messages that contain any of the host names stored in the `$myhosts` variable. These host names would be the first 50 hosts that have performance metrics collected in the `enterprise.com` domain.

Chapter 7: Troubleshooting AQL Queries

Use the information in this section to help you develop and troubleshoot your AQL queries.

Introduction

When composing or trouble-shooting an AQL query, always decompose the query as much as possible to isolate the problem.

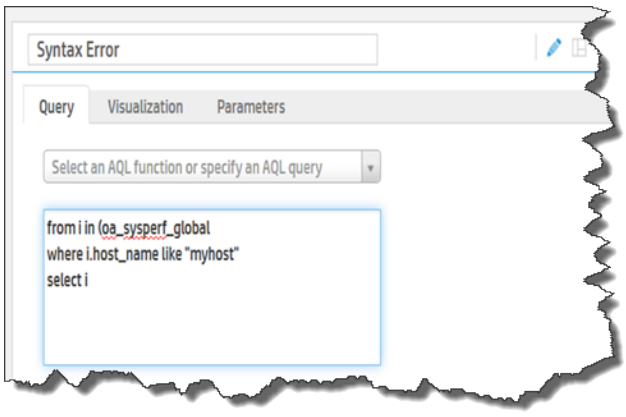
Use the following guidelines to isolate the problem:

- Reduce an AQL list of queries to a single query that is causing the problem.
- Separate the `do` query from the `filter by` query if the original query is a composition.
- Simplify the `where` clause as much as possible to get the query to work.
- Simplify the `select` clause as much as possible to get the query to work.

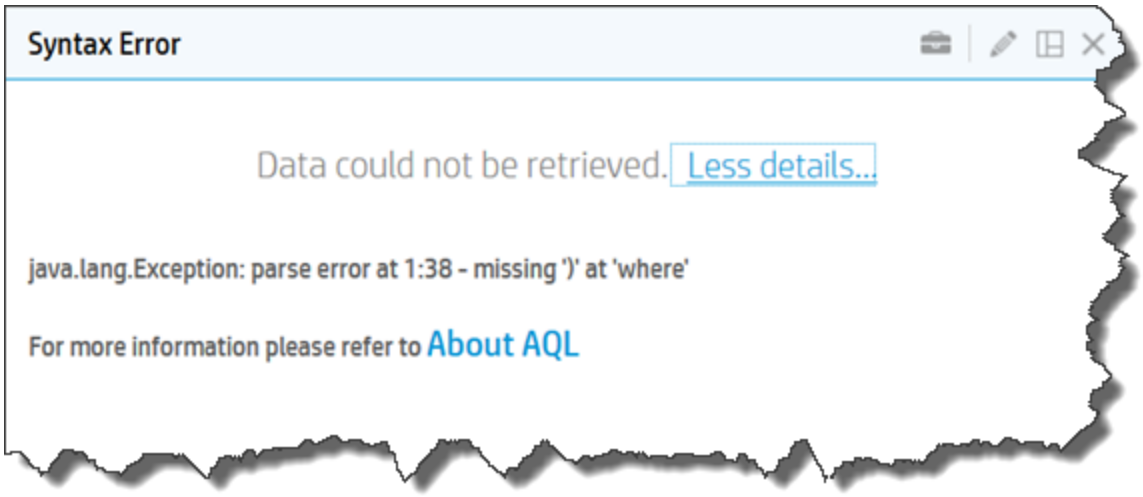
After you reduce the query to a query that works and that returns results, begin adding the removed portions to isolate the problem. Usually the problem will become apparent when you follow this process.

Syntax Errors

AQL queries report syntax errors as 'line:column' where the syntax error occurs. Suppose you created an AQL query with a syntax error in the `from clause` (missing parenthesis) as follows:

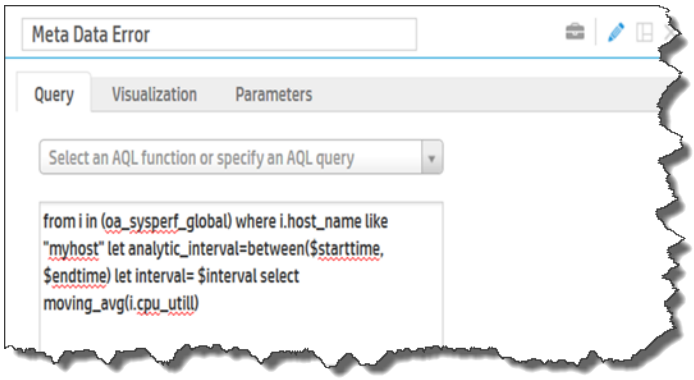


After you run this AQL query, it displays the following error pane indicating the location of the syntax error.

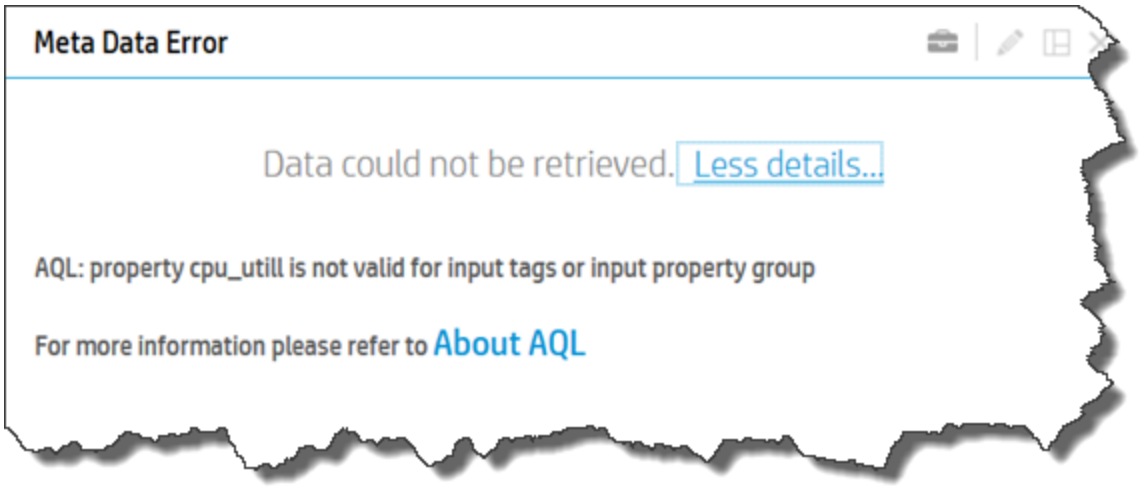


Meta Data Errors

AQL queries report meta data errors as a general error. Suppose you have an AQL query with a reference to data that is not defined (type `cpu_utill` instead of `cpu_util`).



After you run this AQL query, it displays the following error pane indicating the meta data error.



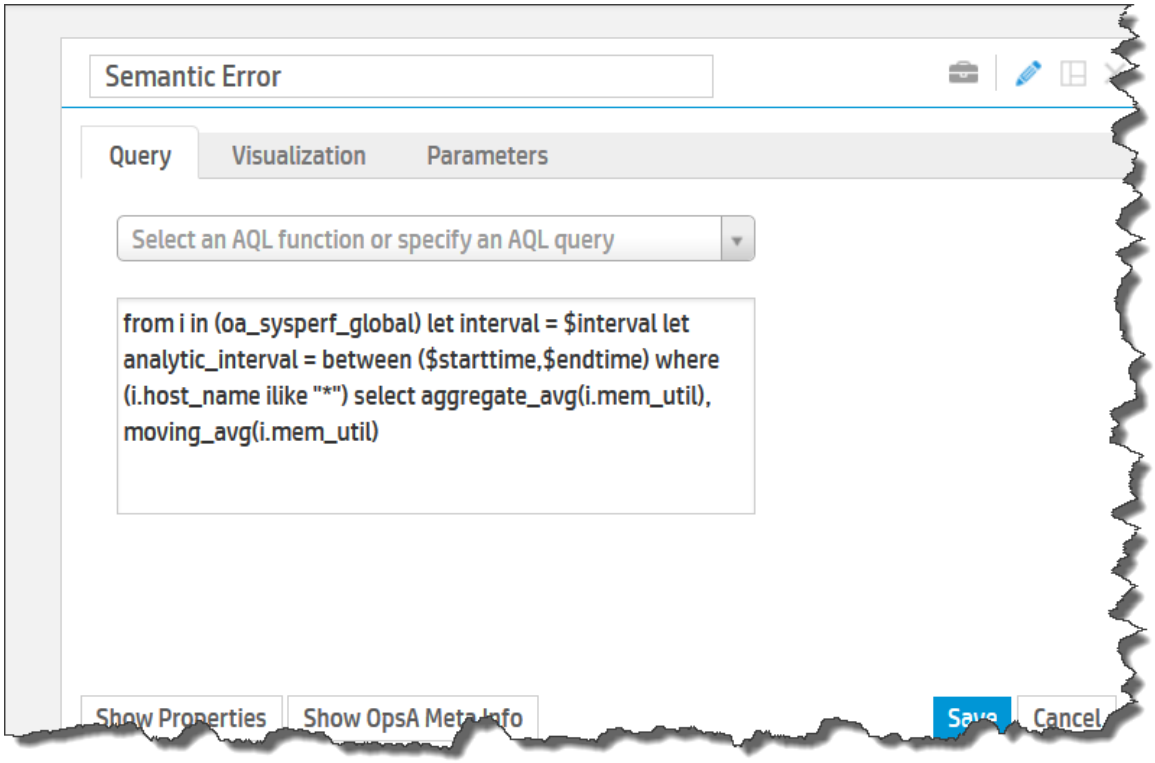
Below are a few other common meta data errors:

- The collection does not exist
- Trying to perform numeric operations on non-metric data types
- Selecting raw metric data with no grouping or operations

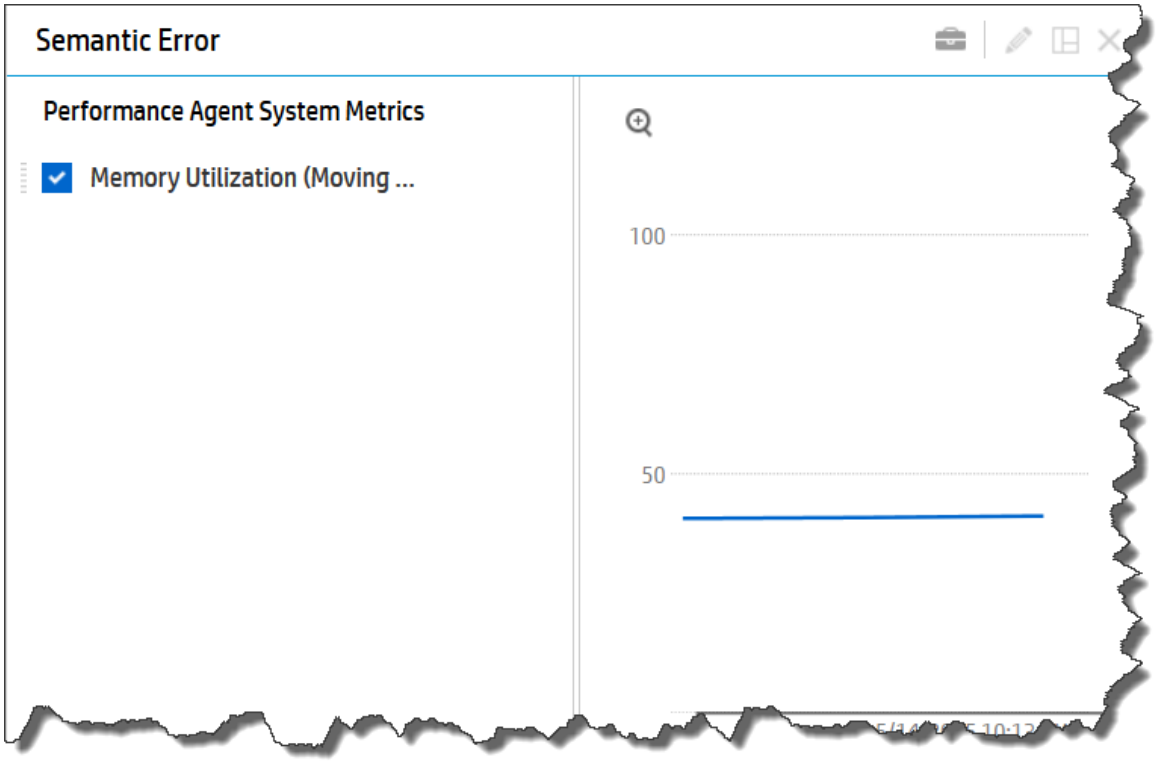
Semantic Errors

There are many potential semantic errors you might encounter when developing an AQL query. A semantic error means the resulting data cannot be properly rendered. For example, time-series data cannot be mixed with attribute value data. Time-series data needs to be rendered as a line chart or heat map and attribute values are typically rendered as a table (although they can be presented as other charts, such as pie charts).

Suppose you have a query in which the select list is mixing time-series data and attribute value data. Note that the `moving_avg` is returning time-series data and `aggregate_avg` is a value.



If you run the above AQL query, then only the time-series (`moving_avg`) data will be displayed as shown in the following graphic; the aggregate value will not be displayed.



When working with AQL queries, the AQL queries you develop will often yield unexpected results. The best troubleshooting technique is to decompose the query as much as possible to make certain that parts of the query are working. It might be that the query is returning disparate data types that cannot all be rendered into a single pane. In such cases you can separate the decomposed query into multiple panes.

Chapter 8: Using R with AQL

The purpose of this chapter is to document the steps that custom analytics developers can take to register custom analytics written using R and make use of them on data being collected by Operations Analytics. Using Operations Analytics 2.10 or newer, Operations Analytics users can run R functions on results from underlying basic AQL functions or expressions that fetch entities and some measurements done for them based on data collected by Operations Analytics. See "Analytics Query Language Functions and Expressions" on page 20 for more information.

Setting up the R Language Pack from Vertica

Operations Analytics uses Vertica's R language runtime environment as the runtime environment for any R function you register with both Operations Analytics and Vertica. It is mandatory that you have the Vertica R Language Pack set up on each node of the Vertica cluster used by your Operations Analytics deployment. You must install the following packages on each node of the Vertica cluster to set up the Vertica R language pack:

- The `compat-libgfortran` package (if required in your version of Vertica).
- The three vertica-R-lang packages.

To install these packages, complete the instructions shown in the *Approach 2: Operations Analytics-Related Extensions* section of the *Operations Analytics Installation Guide*.

Creating the R Functions that Integrate with Operations Analytics

Operations Analytics expects all R functions to conform to the Vertica R UDX framework (R UDX). In order to have a valid R UDX, Vertica expects the following:

1. R functions must have a corresponding UDX factory function written in R. This function must capture input, output frame descriptions, and descriptions of optional input parameters to the core R function.
2. If an R function's output frame does not contain a fixed number of columns with fixed types, then the factory function needs to specify an output type callback R function that is written by the user. The output callback function describes the output frame structure to Vertica at runtime.

3. It is expected that a single .R file is created that contains all of the following:

    a. The UDX factory R function.

    b. Any optional output callback R function.

    c. Any optional parameter callback R function.

    d. The core R function containing the analytics logic or a wrapper function that invokes the analytic R function. This function is basically the main entry point from AQL into custom analytics.

All of the above mentioned pieces of code must be present in a single .R file that is used for registering the R function as a valid Vertica R UDX.

Operations Analytics provides some example .R files containing core R functions, their UDX factory R functions, and output callback functions in the following location:
`/opt/HP/opsa/inventory/lib/hp/r-udx-examples`

See the example named `MVCorr.R` that attempts to do statistical correlation between pairs of time series measurements.

The following snippet from the `MVCorr.R` example demonstrates the boiler plate code that needs to be written to establish the contract with Vertica for the outgoing result or output frame columns. If you want the frame columns output by specific names or want to specify specialized types for some of these columns, you must code the `outtypecallback` R function and register the same in UDX factory R function.

```
mvCorrOutType<-function(x){
ret <- data.frame(datatype=rep(NA,5),lenth=rep(NA,5),scale=rep(NA,5),name=rep
(NA,5))
ret[1,4]="entity"
ret[2,4]="measurement"
ret[3,4]="correlatedentity"
ret[4,4]="correlatedmeasure"
ret[5,4]="correlationcoeff"
ret[1,1]="varchar"
ret[2,1]="varchar"
ret[3,1]="varchar"
ret[4,1]="varchar"
ret[5,1]="float"
ret[1,2]=x[2,2]
ret[3,2]=x[2,2]
ret
}
```

Note: Notice how the input parameters are used by the `mvCorrOutType` callback function to describe the output column names and types.

The names used above in the `outtypecallback` function are directly processed by AQL in its result processing and sent to the dashboard pane in the Operations Analytics console.

The following snippet from the `MVCorr.R` example, illustrates how to write the UDX factory function:

```
mvCorrFactory<-function(){
list(name=mvCorr,udxtype=c("transform"),intype=c("any"), outtype=c("any"),
outtypecallback=mvCorrOutType)
}
```

The following snippet from the `MVCorr.R` example illustrates how to write the main entry point into the custom analytics, possibly as a wrapper function:

```
mvCorr <- function(x){
rvs<-buildRVs(x)
rvObservations<-buildRVObservations(x,rvs)
correlationCoeffs<-buildMVCoefficients(rvObservations)
rvPairsAndCoeffs<-buildRVPairsAndCoeffs(rvs,correlationCoeffs)
rvPairsAndCoeffs
}
```

Identifying the Distinct Time Series Measurements in an Input Frame for an R function

An R function's integration with Operations Analytics currently assumes that the R function is written so that it first identifies the time series measurements (the observations of a metric or measurement at various equally spaced time intervals) in the Operations Analytics domain from the input data frame that is fed to the R function at runtime. The following information helps you better understand the concept of these time series variables and how to write R code to identify these variables in the input frames.

As noted earlier, one can use Operations Analytics dashboard panes to invoke R functions on an AQL function or expression that results in Operations Analytics timeseries data.

At runtime, an AQL function or expression is translated to Vertica SQL statements. When an R function is invoked using AQL on top of an AQL function or expression, AQL additionally wraps these Vertica SQL statements inside of another Vertica SQL statement involving the registered R UDX invocation.

The inner Vertica SQL statement translated from the AQL function or expressions represents the query that Vertica would run internally to supply the results of the same as an input data frame to the R function.

After an entity, its measurements, and their corresponding time series data are identified, each entity and measurement combination could be considered a valid unique instance of a variable backed by the time series data being the observations for the variable.

The following snippet of code from the `MVCorr.R` example demonstrates one way to capture the time series measurement variables before doing either of the following:

- Supplying the pairs of such time series measurements to the core R function.
- Evaluating the correlation coefficient for determining the level of correlation between the pair of measurement variables in question.

```
#
# MultiVar correlation function R UDX entry point
#
mvCorr <- function(x){
rvs<-buildRVs(x)
rvObservations<-buildRVObservations(x,rvs)
correlationCoeffs<-buildMVCoefficients(rvObservations)
rvPairsAndCoeffs<-buildRVPairsAndCoeffs(rvs,correlationCoeffs)
rvPairsAndCoeffs
}
# identify unique combinations of entities and measurements for which to collect
# the time series observations
buildRVs<-function(x){
unique(x[,2:3])
}
#accumulate time series measurements for each combination of entity and
#measurements, thus creating the unique variables under consideration
buildRVObservations <- function(x, rvs){
nRVs <- nrow(rvs)
rvmap<-new.env(hash=TRUE,size=nRVs)
for ( i in 1:nRVs ){
assign(paste(rvs[i,1],rvs[i,2],sep=""),value=i,envir=rvmap)
}
rows <- nrow(x)
tsColumn <- 1
mvSamples <- array(,dim=c(nRVs,0))
ts = x[1,tsColumn]
i = 1
while ( i <= rows)
{
ts = x[i,tsColumn]
colSample <- array(NA, dim=c(nRVs,1))
while ((i <= rows) && (x[i,tsColumn] == ts)){
rvkeytolookup<-paste(x[i,2],x[i,3],sep="")
if ( ! is.null(rvmap[[rvkeytolookup]]) ){
colSample[rvmap[[rvkeytolookup]]] = x[i,4]
}
i <- i + 1
}
mvSamples <- cbind(mvSamples,colSample)
}
mvSamples
}
```

```
# Iterate through list of variables and invoke R core function cor to calculate
correlation coefficient between #each unique pair of variables
buildMVCoefficients <- function(multiVarSamples) {
nRandomVars <- nrow(multiVarSamples)
multiVarCorCoef <- array(0, dim=c(nRandomVars, nRandomVars))
for (i in 1:(nRandomVars-1)) {
for (j in (i+1):nRandomVars) {
multiVarCorCoef[i,j] <- cor(multiVarSamples[i,], multiVarSamples[j,], use =
"na.or.complete")
}
}
multiVarCorCoef
}
# buld final results to be returned to caller of R UDX.
buildRVPairsAndCoeffs <- function(rvs,rvCoeffs) {
entity<-c()
entitymeasure<-c()
correlatedentity<-c()
correlatedentitymeasure<-c()
correlationcoefficient<-c()
for ( i in 1:(nrow(rvs)-1) ) {
for ( j in (i+1): (nrow(rvs) ) ) {
entity<-c(entity, as.character(rvs[i,1]))
entitymeasure<-c(entitymeasure, as.character(rvs[i,2]))
correlatedentity<-c(correlatedentity, as.character(rvs[j,1]))
correlatedentitymeasure<-c(correlatedentitymeasure, as.character(rvs[j,2]))
correlationcoefficient<-c(correlationcoefficient, rvCoeffs[i,j])
}
}
result <- data.frame
(entity,entitymeasure,correlatedentity,correlatedentitymeasure,correlationcoefficie
nt)
result
}
```

Registering an R Function

You must register a newly created R Function with both Vertica and Operations Analytics.

Registering your R function with Vertica

1. Prepare the `.R` file so that it contains the following:
   - The core R function implementing your custom analytics logic or wrapper function that actually calls your core custom analytics.
   - The Vertica R UDX factory function.
   - Output type callback R function.
   - Any other helper R functions used by the core R function.
2. Run the Vertica R UDX load commands to load the R function into Vertica. At this stage the R function becomes available as a valid UDX that can be invoked from Vertica SQL.

Note: You must complete these steps as a valid Vertica database user who has the privileges to run SQL commands and who can create UDX functions in the Vertica database system.

The following is an example of the pair of Vertica SQL commands required to load the example R UDX provided in the `MVCorr.R` example:

```
create library mvCorrLib as '/home/dbadmin/functions/MVCorr.R' language 'R';
create transform function mvCorr as language 'R' name 'mvCorrFactory' library
mvCorrLib;
```

You can also review the Vertica documentation about how to load Vertica R UDX functions.

Registering your R function with Operations Analytics

After the R function is loaded and available in Vertica, you must tell Operations Analytics about it by registering an R function module into Operations Analytics.

1. Create an R module specification file. The following example shows the contents of one such module definition file that defines the R module for the multi-variate correlation R UDX function example from the `/opt/HP/opsa/inventory/lib/hp/r-udx-examples/mvCorr.R` file.

   ```
   module MultiVariate;
   /* Does multivariate correlation */
   define mvCorr input(any, integer, integer) output(any);
   ```

   Save the content in a text file. For example, see `/opt/HP/opsa/inventory/lib/hp/r/multivariate.rpsec`
2. Load the R module specification into OPERATIONS ANALYTICS by running the following command:
   `/opt/HP/opsa/bin/opsa-rspec-module-manager.sh -?`

You should see an output similar to the following:

```
OPSA_HOME is set to /opt/HP/ops

-t <tenant name> Name of Tenant (mandatory argument except when using –v
option)
-v <file> Validate File
-l modules List Summary of Loaded Modules
-l all List Contents of All Loaded Modules
-l <modulename> List Contents of Module
-i <file> Import File
-a <authorname> Specify Author for Import File
-d <modulename> Delete Module
-? This help message
```

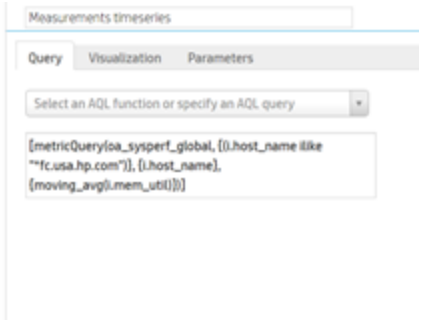
For example, you could load the R module named `MultiVariate` previously defined in the `/opt/HP/opsa/inventory/lib/hp/r/multivariate.rpsec` file by running the following command:

```
/opt/HP/opsa/bin/opsa-rspec-module-manager.sh – t opsa_default -i
/opt/HP/opsa/inventory/lib/hp/r/multivariate.rpsec
```

Using your R Function in an `Operations Analytics` Dashboard

You can create a dashboard pane with your AQL function or expression that returns time series data that you can visualize using an Operations Analytics line chart, heat map chart, or bar chart elements in the Operations Analytics console.

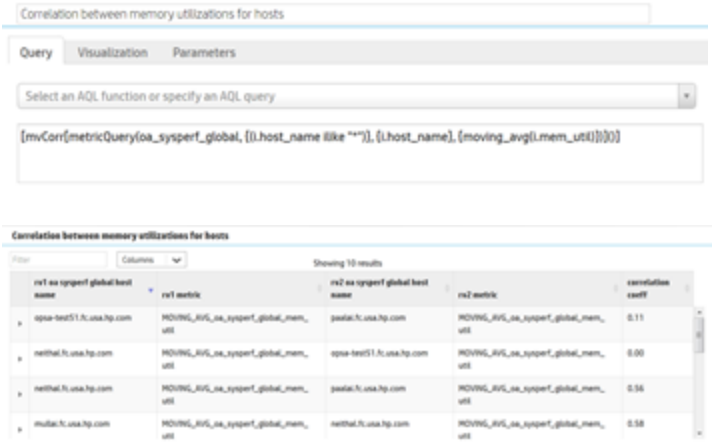
In a dashboard pane, you can visualize the results of the AQL function by using it in the query edit box for the pane as shown below:



The query used in the query editor for the above dashboard pane is:

```
[metricQuery(oa_sysperf_global,{i.host_name ilike "*<mylocation>.<mycompany>.com"},
{i.host_name}, {moving_avg(i.mem_util)})]
```

Now surround the AQL function call with a call to a registered R function as shown below to trigger the invocation of the registered R function:



The query, after surrounding the AQL function call with the invocation of the registered R function, looks as follows:

```
[mvCorr[metricQuery(oa_sysperf_global,{i.host_name ilike
"*<mylocation>.<mycompany>.com"},{i.host_name}, {moving_avg(i.mem_util)})]()]
```

Limitations

Only a table visualization of the invoked R function is supported in Operations Analytics 2.20 or newer.

Send Documentation Feedback

If you have comments about this document, you can contact the documentation team by email. If an email client is configured on this system, click the link above and an email window opens with the following information in the subject line:

Feedback on AQL Developer Guide (Operations Analytics 2.31)

Just add your feedback to the email and click send.

If no email client is available, copy the information above to a new message in a web mail client, and send your feedback to sw-doc@hpe.com .

We appreciate your feedback!